1983

Utilization of Ada as a program design language

Wylie, George J.; Watt, Thomas R.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/20028

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

UTILIZATION OF ADA AS A PROGRAM DESIGN LANGUAGE

by

George J. Wylie

and

Thomas R. Watt

June 1983

Thesis Advisor: Ronald Modes

Approved for public release, distribution unlimited

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Utilization of Ada as a Program Design Language

5. TYPE OF REPORT & PERIOD COVERED
Master's Thesis
June 1983

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
George J. Wylie and Thomas R. Watt

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93940

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93940

12. REPORT DATE
June 1983

13. NUMBER OF PAGES
110

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release, distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Ada, PDL, SDL, Software Engineering

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In terms of manpower, time and money, the single largest investment that must be made in the acquisition and maintenance of a large and complex computer system is the investment made in software. In response to this situation, the DOD began an intensive and comprehensive research and development effort in an attempt to reduce, if not eliminate the inherent problems associated with software system design. The end result of this effort was the creation of the Ada programming language. This thesis will examine the development

BLOCK 20: ABSTRACT (Continued)

of the language, focusing attention on the concepts and features which make Ada a potential "software crisis" solution. These concepts and features will be further examined as to the extent to which they support the utilization of Ada as a program design language (PDL).

Approved for public release, distribution unlimited

Utilization of Ada as a Program Design Language

by

George J. Wylie
Lieutenant Commander, United States Navy
B.A., University of Washington, 1971

and

Thomas R. Watt
Lieutenant, United States Navy
B.A., Syracuse University, 1973

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
June 1983

ABSTRACT

In terms of manpower, time and money, the single largest investment that must be made in the acquisition and maintenance of a large and complex computer system is the investment made in software. In response to this situation, the DOD began an intensive and comprehensive research and development effort in an attempt to reduce, if not eliminate the inherent problems associated with software system design. The end result of this effort was the creation of the Ada programming language. This thesis will examine the development of the language, focusing attention on the concepts and features which make Ada a potential "software crisis" solution. These concepts and features will be further examined as to the extent to which they support the utilization of Ada as a program design language (PDL).

TABLE OF CONTENTS

I. INTRODUCTION: BRIEF HISTORY OF THE DEVELOPMENT OF ADA

A. BACKGROUND: THE SOFTWARE CRISIS

In terms of manpower, time and money, the single largest investment that must be made in the acquisition and maintenance of a large and complex computer system is the investment made in software. During the last thirty years the cost per executed instruction of computer hardware has declined by a factor of two every two to three years; while the relative cost of software has increased dramatically, from under 20% of total computing costs in 1960 to cover 80% of those costs in 1980. The increasing ratio of software to hardware costs is most acutely demonstrated in complex embedded computer systems developed for and used by the Department of Defense. In 1973 over 50% of the DOD's total software expenditures were dedicated to embedded systems, and today that figure is significantly higher. Coupled with this is the fact that complex embedded software projects have frequently experienced substantial cost and schedule overruns and have sometimes had to be abandoned altogether because their sheer complexity resulted in project attempts which became altogether unmanageable [Ref. 1: p. 6].

The term "software crisis" was coined in the late sixties to describe what was becoming a wholly untenable situation. Thousands of programmers and analysts using hundreds of

languages for use on hundreds of computers with untold variations in applications resulted in an overall software picture within DOD that was simply beyond comprehension, let alone management. In response to this situation, the DOD began in 1975 what was to become one of the most comprehensive and forward looking programs in the history of software engineering--the development of the Ada language [Ref. 2].

Prior to 1975 it was realized within DOD that one of the greatest problems afflicting the management and use of computer software was that there were simply too many different software languages in use (roughly 400, counting all languages and dialects in use at that time). The great diversity of languages in use, coupled with the wide variety of applications required by DOD embedded software systems, resulted in the fact that portability of software between systems was in most cases impossible. Additionally, since DOD embedded systems tended to be very complex, each system tended to become an island unto itself regarding the system's acquisition, maintenance and personnel support resulting in the need for specialized tools and personnel training for each system.

B. THE HIGH ORDER LANGUAGE WORKING GROUP

The multiple language problem begged as its logical panacea the formulation or adoption of a single software

language which could be utilized on all embedded system software applications. The DOD [illegible] reasoned that ~~if~~ a universal language could promise ~~be~~ adopted, substantial gains in the control over such problems as non-portability, extensive maintenance and programmer training expenditures, and software understandability ~~could be~~ achieved. To address the problem of multiple languages, as well as to attempt a reasonable solution to that problem, the High Order Language Working Group (HOLWG) was formed in 1975 with representatives from the Army, Navy, Air Force, Marines and other Defense agencies. The HOLWG's charter was to explore and identify the DOD's requirements for computer programming languages, evaluate existing languages then being used by the DOD, and finally to recommend the implementation and control of a "minimal set" of languages for use throughout the DOD. During 1975 and 1976, HOLWG undertook an extensive requirements analysis process in accordance with its charter, beginning with the publication of STRAWMAN, which was essentially a questionnaire with which to stimulate comments from the field. In August of 1975 the WOODENMAN document was written which summarized the comments and recommendations received through STRAWMAN. Further solicitation of comments from worldwide sources followed, and the results were again analyzed, leading up to the publication of TINMAN, which was

a complete set of requirements for the intended universal language.

The research up to this point had revealed that though the differences in embedded system software applications were substantial, the programming language requirements for a broad spectrum of those applications were remarkably similar. It was, for instance, clear that for all applications, such programming methodology attributes as top-down design, structured programming and information hiding were desirable features to utilize as these features enhanced management's ability to improve programmer productivity, system reliability and overall system control. In addition these features were seen as essential toward making possible the development of advanced programming tools with the potential to significantly improve the productivity of DOD software engineers.

The publication of TINMAN in January of 1976 was followed by an intensive examination of existing languages, and a formal evaluation of those languages against the requirements spelled out in TINMAN. As might be expected, no single existing language was found acceptable as meeting those requirements. The primary reason for this is that each language was irretrievably entangled within the application it supported, and in most cases the different languages and

dialects were initially designed for a specific application, and only later applied to a broader applications base.

Although no existing language was recognized as being capable of adoption as a universal DOD computer language, the HOLWG did recognize an immediate need to stop the proliferation of newer, though technically similar languages in the acquisition and maintenance of embedded system software projects. In April of 1976 the DOD released Directive 5000.29, which restricted all projects to "DOD approved high order programming languages" unless cost effectiveness or technical practicality was significantly impaired over the system's life cycle in complying with that directive. More specificity was offered by DOD Directive 5000.31 in November, 1976, in that this directive specifically listed the languages authorized for use by the DOD. Those languages are FORTRAN and COBOL (DOD), TACPOL (Army), CMS-2 and SPL/1 (Navy), and JOVIAL (Air Force). The issuance of these directives was intended only as an interim measure rather than a long term solution to the underlying problem of too many embedded system computer languages, and as such the directives served only to thwart the development of additional and assumably unnecessary new programming languages. Given that a considerable amount of investment had already taken place in these approved languages, there was little immediate need to replace them in their present applications.

C. THE NEED FOR A SINGLE PROGRAMMING LANGUAGE

But the basic questions still remained: was it feasible to develop a universal language, and of the many languages in use both within and outside the DOD , which if any language would best serve as a model to emulate in the development of a new language. In answering the first question the DOD performed two independent cost benefit analyses, both of which concluded that it was appropriate to undertake the development of a new universal language which fully met the TINMAN requirements. The ultimate benefits possible from such an undertaking would most likely range in the hundreds of millions of dollars in personnel training savings, and savings realized through greater use of compilers and other software tools. Toward an answer to the second question, the HOLWG was tasked with executing the development of a common language, while program management responsibility rested with the Defense Advanced Research Projects Agency (DARPA). The criteria imposed on the HOLWG were that it develop a language commensurate with state-of-the-art technology and design methodologies and that it develop a language of high enough quality so as to be attractive to interests outside the defense industry, including (it was hoped) industry, universities, foreign vendors and NATO allies.

In January, 1977, ~~the~~ IRONMAN document was published as a requirements definition for the new common language, and this

document served as the criterion for an international competition against the Request For Proposal issued in April of that year. Seventeen vendors responded to the RFP, from which four were selected to proceed with further design. All four vendors indicated an intention to use the programming language PASCAL as a starting point in their respective development efforts. Preliminary design efforts were to be measured against the REVISED IRONMAN requirements definition document, which was released in July of 1977. Evaluation of the preliminary designs was accomplished by distributing the preliminary designs among approximately eight DOD and non-DOD evaluation teams from the United States, Europe and the United Kingdom. This Phase 1 evaluation resulted in the selection of two of the four vendors for continued development, and in June of 1978 the final requirements document, STEELMAN, was issued. In March of 1979 the final designs were issued from the two competing vendors, CII-Honeywell Bull and Intermetrics.

Again the proposed designs were distributed among evaluation teams around the world for comment, and software engineers from many disciplines were invited to meet with and question the designers to better understand their design rationale. The results of these meetings, the comments received from the evaluation teams, and an intensive analysis

by the different DOD interests resulted in the selection of CII-Honeywell Bull in April of 1979.

Up to this time no official name had been given the new language, though the industry press had unofficially dubbed the name DOD-1. HOLWG objected to the inclusion of any reference to the Department of Defense in naming the new language, as such reference could have the effect of discouraging acceptance and use of the language in the non-military marketplace, and one essential objective of HOLWG was to specifically enhance the possibility of acceptance in that marketplace. The language name Ada was adopted in the Spring of 1979 in honor of Agusta Ada Byron, famous in her role as the first "computer" programmer.

In June of 1978 the SANDMAN document was issued, which addressed the need to develop an integrated system of software development and maintenance tools along with development of the language itself. HOLWG reasoned that though no special environment would be needed to use Ada, the acceptance of the language and the potential benefits possible from the development of the language could be greatly enhanced with the implementation of a standardized programming environment. SANDMAN was reviewed as to its intent and possible alternatives at a workshop jointly sponsored by the Army, Air Force, Navy and University of California at Irvine in late June, and this workshop resulted

in the preliminary draft statement of Ada environmental requirements, PEBBLEMAN, in July, 1978. This document described all aspects of the Ada environment, including language standards, policy, configuration control, compiler validation and software and management tools. After wide dissemination for comments as to its contents, PEBBLEMAN was revised in January, 1979, to reflect the concerns mentioned by those submitting comments.

D. THE ADA PROGRAMMING SUPPORT ENVIRONMENT

A set of technical requirements for what was designated the Ada Program Support Environment (APSE) was published and distributed in November, 1979, as Preliminary STONEMAN, along with invitations to selected interests to attend an APSE workshop on 27-29 November, 1979, in San Diego. At this workshop, two hundred and twenty (220) industrial, research, academic and government participants contributed opinions and recommendations as to the APSE, and the results of the workshop were reflected in the final STONEMAN document which was published in February 1980. The STONEMAN requirements document delineated the structure and content of necessary elements to an embedded Ada system, including the support system of the host machine and the run-time system on the target machine. Considerable emphasis in STONEMAN is placed on the standardization and coordination of a well defined set of tools and uniform interfaces within the APSE to enhance

Ada programming support throughout its life cycle. Such uniform conventions, with Ada used to implement the APSE, would also enhance such desirble system attributes as portability, modularity, uniformability and understandability. In addition to the APSE, STONEMAN delineated the Kernel Ada Programming Support Environment (KAPSE) to ensure standardization of the environment made available to the APSE (and therefore ensure APSE portability) in the event more than one APSE evolved. STONEMAN also defined a Minimal Ada Program Support Environment (MAPSE) as a minimum set of functions which an APSE should perform. As defined by the MAPSE, the minimal APSE should be able to create database objects, produce new objects which are records of analysis of other objects, transform objects from one representation to another, support object display, parse, link, load and execute.

In addition to development of the Ada language and the support environment within which it will reside, the DOD is encouraging and funding the development of compilers to interface the Ada language with the intended object machines on which the language code will be processed. Individual efforts are being made by each of the DOD branches so as to best match branch (Army, Navy, Air Force) needs with the overall development of the Ada language. Perhaps more importantly, the DOD wants to encourage the use of Ada for

exploratory development and advanced development projects even before the military service and accredited industry production quality compilers are available. As an example, the Defense Advanced Research Projects Agency (DARPA) funded Intermetrics Corporation (the vendor eliminated during the Phase 2 language development vendor selection process) to develop a test compiler to run on the DEC-20, using the test translator it had developed during the Phase 2 language design effort. Though the formal intent (and investment) of the DOD is to develop compilers and other tools which will serve the needs of the developers and maintainers of software for execution on military computers, there is a recognized, though less formal, intent to make available to commercial vendors the compiler products developed under the military umbrella. This attitude is consistent with one of the initial criteria imposed by the DOD on HOLWG; that of creating a language of high enough quality so as to be attractive to commercial and other interests outside the military arena. To this end, the DOD has attempted to enhance military-industrial communication by offering compilers, programming tools and compiler test sets upon their completed development to commercial vendors, with the provisoN that such vendors pay a nominal fee for distribution and submit trouble reports to the DOD when trouble with the unit is recognized. In addition, Ada software developed by

the DOD (except that software belonging to the core Ada development and introduction program) will be available to commercial vendors on an unlimited basis, unless it is determined that procurement by the vendor under limited rights (and DOD's right to redistribute) would result in significant savings to the government. It is reasoned by the DOD that encouraging interaction between the military and non-military interests involved or potentially involved in the Ada language and its surrounding environments and tools, will foster the growth and acceptance of Ada in the non-military environment.

E. EDUCATION IN THE ADA LANGUAGE

Commensurate with the development of any new programming language is the need to address the education of those who will be responsible for the development and maintenance of that new language. This need is brought into even greater focus with Ada, since Ada incorporates new concepts and facilities as yet unseen in prior programming languages. To begin with, Ada is perhaps the first programming language where the system goals of modifiability, efficiency, reliability and understandability were specifically and formally recognized as necessary goals of the language prior to the initial design of the language. In support of these goals, the software design principles of modularity, abstraction, information hiding, localization, uniformity, completeness

and confirmability were also specifically recognized as necessary elements to the language prior to its design.

With these design goals and principles established as a backdrop to the Ada design effort, certain specific and, in most cases, unique design characteristics evolved out of the Ada design endeavor. These characteristics include an object-oriented design methodology, strong type-checking across module boundaries, packages for specifying logically related collections of resources, high-level concurrent programming facilities, tasking to permit communicating sequential processes, and a system framework wherein the language itself and the support environment within which it resides are seen as a "unit".

Together, these goals, principles and design characteristics provide a novel training opportunity to present a coordinated view of modern programming practices, and to this end, the HOLWG established a Subcommittee on Education and Training in March of 1979. The guiding philosophy of the Subcommittee on Education and Training has been that Ada represents the cutting edge of a new software technology that will inevitably result in a more structured working environment for programmers and program managers, and an environment that will require the adoption of interface conventions which will, to a large part, rely on modules developed by others. It is not enough to accomplish local

changes in existing language programs if the proper use of Ada's novel features are to be realized, though spinoff courses such as "Ada for FORTRAN programmers" or "Ada for Pascal programmers" will undoubtedly appear. Rather, Ada requires the programmer to internalize a top-down, modular approach to program design that results in programs whose structure is substantially different from existing high level or assembly language programs. The new style of modular thinking required for the effective design and use of Ada programs is more difficult to teach than the syntax and semantics of the language itself. This is in large part due to the fact that many of the issues involved in teaching the Ada design methodology are language independent; while the Ada language provides linguistic support for the modern software technology on which it is based, the underlying software methodology and problem-solving techniques are themselves independent of Ada or any other particular language. This "independency" between language and software methodology was not by accident. Rather, it was specifically intended by the HOLWG as a means to more easily accomplish the possible transition from Ada to whatever future languages that might come along. The HOLWG reasoned that though languages may come and go, a sound and well designed system of underlying software methodology and problem solving techniques will stand a better chance of survival over time

and will provide a standard and foundational basis with which to describe the desirable properties of programming languages in general, as well as a basis from which future programming languages in particular can be developed.

The threshold, then, to be overcome in the training of programmers and program managers in the Ada language is substantially higher than that of previous languages, as Ada demands an understanding of its underlying philosophy and approach prior to the understanding of its syntax. This is truly a "macroscopic" approach to language design, since it forces the programmers and program managers to maintain a perspective on the language which sees the language as merely a vehicle with which to enforce the far more important philosophy behind the language and the goals and principles supporting the language. This approach has the added benefit of forcing the designers and users of Ada to consider the effects of using Ada as a programming language for any specific project over the project's entire life cycle rather than in piecemeal fashion.

To meet the Ada training challenge, the Subcommittee on Education and Training is coordinating various individual and coordinated training programs among the components of the DOD and, in addition, is endorsing education and training endeavors within the commercial and academic sectors both at home and abroad.

F. ADA AS A "SOLUTION" TO THE SOFTWARE CRISIS

There is little doubt that the development of Ada has caused considerable commotion in the computer software arena. Some would argue that the horse is finally once again ahead of the cart in software development in that with Ada there seems to have been no expense spared in laying a comprehensive and foundational design strategy and framework prior to designing the language syntax itself. Ada appears to represent for the first time an attempt to build a fundamentally new software design philosophy rather than just another "new" but, in fact, patchwork software language. The software crisis is with us and will remain so for many years to come, but the injection of the Ada language into that crisis will at least slow what is now an uncontrollable rate of growth of that crisis. This, of course, assumes that both DOD and non-DOD interests continue with their efforts on what appears to be the right path.

II. SOFTWARE ENGINEERING CONCEPTS UTILIZED IN THE DEVELOPMENT OF ADA

The fundamental reason for the existence of the "software crisis" is due to the unmanageable complexity of software systems in general. As tools for software development improve and as software system design experience increases, this situation is clearly becoming more difficult to deal with as newer and greater problems arise. A solution for reducing the complexity of software systems is attainable through close adherence to the goals and methodologies of software engineering supported by a high order language that promotes and enforces these principles.

Software engineering is modeled on the techniques, methods and controls associated with hardware development. Although fundamental differences do exist between hardware and software, the concepts associated with planning, development, review, and management control are similar for both system elements. The key objectives of software engineering are (1) a well defined methodology that addresses a software life cycle of planning, development and maintenance; (2) an established set of software components that documents each step in the life cycle and shows traceability from step to step; and (3) a set of predictable milestones that can be

reviewed at regular intervals throughout the software life cycle [Ref. 3: p. 15].

The purpose of this chapter is to delineate the goals of software engineering and discuss the associated principles that enable software system designers to attain them [Ref. 4]. This chapter will also discuss software development techniques and tools that utilize these software engineering principles in the design of software systems.

A. THE GOALS OF SOFTWARE ENGINEERING

The most fundamental goal in the design of software is to ensure that the resultant product satisfies the designated requirements. Unfortunately, there often arises a misinterpretation of the user's stated requirements by the system implementers. As a result of this misunderstanding, changes in requirements during the life cycle of a software system is inevitable.

The acceptance of the inevitability of changes in requirements during software development has resulted in the establishment of a set of goals that overcome the effects of such change. Four properties that are sufficiently general to be accepted as goals for the entire discipline of software engineering are modifiability, efficiency, reliability, and understandability.

1. Modifiability

The goal of modifiability is the most difficult goal to master and to measure. Modifiability implies controlled change, in which some parts or aspects remain the same while others are altered, all in such a way that a desired new result is obtained. Modification during software development may occur as a result of a change in the system requirements or in response to the correction of an error made earlier during the development phase of the system.

The modification of a system must take into consideration the maintenance of structural integrity. If this consideration is ignored during modification, the software will become segmented, resulting in a potential loss of logical flow. This will invariably lead to the original design becoming vague and unintelligible, making follow-on modification to the system very difficult. The key to system modifiability is that it should promote the ability to change the software without enlarging the complexity of the system.

2. Efficiency

In well engineered systems there is a natural tendency to use critical resources efficiently [Ref. 3: p. 284]. These resources are classified into two basic groups--time and space resources. Time resources are generally concerned with process execution in a predetermined timeframe; hence, they tend to be hardware dependent.

However, selection of the proper software algorithms will obviously enhance the time of execution. Space resources are concerned with the physical side of the execution process.

Embedded systems are often required to consider both classifications to promote efficiency. If the embedded system is concerned with real events, time resource efficiency becomes paramount. If the embedded system is constrained by the physical size of the existing hardware, then space resources become the overriding concern. Most often, the efficient use of the two classifications at the same time is not attainable and a tradeoff must occur.

In order for efficiency to be attained in a system, it must be considered throughout the entire system development and not just in the early phases as is most common. Insights reflecting a more unified understanding of a problem have far more impact on efficiency than any amount of "bit twiddling" within a faulty structure.

3. Reliability

As more and more computer systems are being developed to operate for long periods of time with minimum operator interference, reliability of the system is taking on greater importance as the price of system failure reaches unacceptability. Reliability must both prevent failure in conception, design, and construction, as well as recover from failure in operation or performance.

As with efficiency, reliability must be a concern throughout the entire software development program. Most often reliability is considered too late, or not at all, in most software development efforts. Reliability can only be built in from the start; it cannot be added on at the end. Hence, reliability has a pervasive and crucial effect on software engineering practices. A well engineered, reliable computer system must fail gracefully with little or no effect to the system overall.

4. Understandability

Understandability is the key to the proper management of the complexities inherent to software systems. Understandability is not exclusively a property of legibility. The entire conceptual structure is involved. Understandability bridges the true system with the perceived system. Although understandability is a prerequisite to reliability and modifiability, it is also important as a goal in itself because it draws attention to the complexity of the system. The only way to achieve understandability is to impose clearly notated structure and organization on the system.

System understandability is further enhanced from the impact on the system from various levels in the structure. From the lower levels, proper coding styles lend themselves to understandability. At the higher levels, the ability to

expedite the segregation of various algorithms and data structures aid in understandability attainment.

B. THE PRINCIPLES OF SOFTWARE ENGINEERING

The software engineering goals are clearly applicable to most if not all software systems. These goals, however, are not attainable simply through utilization of any software development methodology. In order to achieve these goals, the software development approach must be highly structured, well disciplined and closely adhere to a basic set of software engineering principles that support these goals. The principles of software engineering include abstractions, information hiding, modularity, localization, uniformity, completeness, and confirmability. Proper utilization of these software engineering principles can result in the development of a software system that is modifiable, efficient, reliable, and understandable.

1. Abstraction

As stated previously, the inability to manage the complexity of software systems is the primary cause of the "software crisis." Abstraction lends itself to managing the complexity. Abstraction exists in varying degrees throughout all levels of the systems hierarchial structures. Each level of abstraction is built from lower levels which in turn were built from even lower levels in the hierarchy. In developing software systems, the level of abstraction that satisfies the

stated requirements is utilized. The essence of abstraction is to extract essential properties while omitting unessential details. The levels of abstraction formed through hierarchial decomposition, display an abstract view of the lower levels purely in the sense that details are subordinated to the lower levels. The principle of abstraction ensures that a given level in a hierarchial decomposition is understandable as a unit, without requiring either knowledge of lower levels of detail or necessarily how it participates in the software system as viewed from a higher level.

2. Information Hiding

Information hiding enforces the abstraction principle. Where abstraction was concerned with the extraction of essential details of a given level, the purpose of information hiding is to make inaccessible certain details that should not affect other parts of a system. Abstraction helps to identify details that should be hidden, while hiding is concerned with defining and enforcing access constraints.

The application of the information hiding principle in conjunction with the abstraction principle promote goal achievement. These two principles, besides encouraging system efficiency, assist in the maintainability and understandability of a software system through reduction in the amount of specific details a system programmer would be requested to know at any particular level. System

reliability is also elevated through the application of these two principles, for at each level only certain predefined operations are permitted to occur preventing any inadvertent operation from taking place that could violate the logical structure of that level.

3. Modularity

It has been stated that modularity is the single attribute of software that allows a program to be intellectually manageable [Ref. 5]. Modularity is concerned with the dividing of a program into subprograms (modules) which can be compiled separately. Modularity yields a hierarchial structure, for when decomposition of the software system occurs levels of program modules are created.

In utilizing a top down approach in software design, a decomposition of each successive level into distinct functional modules will occur. Most often, higher level modules are related to high level abstractions, and therefore are generally machine independent. In addition, a higher level module will specify what action is to be taken, while the lower level modules define how that action is to be carried out. Lower level modules are generally machine dependent. If a bottom-up approach to software design is initiated instead, decomposition of the system begins at the bottom of the hierarchy resulting in the creation of highly complex modules at the top of the system.

The key to the enhancement of system reliability through the use of modularity is to ensure that a well defined interface exists between each module. A well defined interface is an explicit set of assumptions one program module makes about another. These interfaces are the "connections" between modules. A measure of the strength of these interconnections among modules is known as coupling. Loosely coupled modules are most preferred for they result in greater modular independence. Cohesion is another modular measurement which defines how tightly bound or related its internal elements are to one another within the module proper [Ref. 6: p. 85]. Strong cohesion within individual modules is most desirable for it implies that the components of a particular module are functionally and logically dependent.

4. Localization

The principle of localization assists in developing program modules which demonstrate loose coupling and strong cohesion. The principle of localization is concerned with physical proximity where related elements are brought together all in one module resulting in a reduction in resource redundancy. Through the use of the localization principle, logically related items are collected into one physical module, forming a module that exhibits strong cohesion. The localization principle also implies modular independence, resulting in a much desired loosely coupled system.

The principles of localization and modularity lend themselves greatly to the attainment of the goals of software engineering. If a software system is developed through the implementation of these principles, then the understandability of any particular module should be possible independent of the other modules in the system. Subsequently, since these principles tend to localize design decisions into pre-defined modules, the effects of modification to the system can be minimized to a smaller more manageable collection of modules. The goal of system reliability is enhanced due to the fact that the proper use of these principles will result in a reduction in the number of modular interfaces.

5. Uniformity

The uniformity principle is directly related to the software engineering goal of understandability. Uniformity is concerned with intermodule notational consistency in areas such as naming conventions, code structure, interface descriptions, etc. Uniformity is achieved through the use of proper coding techniques, where application of a consistent control structure and calling sequence for operation is utilized and where the depiction of logically related items are identical at any particular level.

6. Completeness

The purpose of this principle is to ensure that all essential elements have been included in the software system development. Completeness is achieved through proper iterative design procedures through the system development phases. Completeness in combination with the abstraction principle, results in the development of necessary and sufficient modules supporting the goal of reliability. Completeness also enhances the efficiency goal, because it becomes possible to adjust a lower level module without affecting modules in the higher levels.

7. Confirmability

The principle of confirmability is concerned with achieving stated goals contained in the software system requirements and specifications. It is, therefore, paramount to ensure that system requirements are accurate and that system specifications are testable. Confirmability implies that decomposition of the software system must occur so that it can be readily tested resulting in a system that is modifiable. This principle is most commonly realized through the use of informal software system reviews such as structured walkthroughs [Ref. 3: p. 141].

C. SOFTWARE DEVELOPMENT DESIGN METHODOLOGIES

Software engineering principles will, when correctly applied, achieve software engineering goals. However, these principles cannot be implemented in a casual, hit or miss fashion. As software systems are becoming more and more modularized, a uniform system decomposition standard must be adhered to.

There are generally four recognized design methodologies that exhibit a uniform standard for system decomposition. These are top down structured design, data structured design, Parnas decomposition criterion and object oriented design.

1. Top Down Structured Design

The top down structured design approach is based upon the hierarchial organization of modules. This approach suggests the decomposition of a system is achieved by making each step in the process a module [Ref. 6: p. 106]. This approach begins with the top level module designed in terms of the modules of the next lower level. In essence, a determination is made as to what type of modules will be required on the next lower level and how they should be connected to form the top level module. As this is happening, no consideration is given about the detailed construction of the second lower level modules until the top level module has been satisfied. This process continues until all modules at all

levels are formulated in terms of the modules below them. This process results in program modules that are highly functional and well defined. The higher level modules contain the highest levels of abstraction, while the lower level modules contain the primitives of the system which implement operation in response to higher level actions.

2. Data Structure Design

The data structure design methodology converts a representation of data structure into a representation of software. Utilizing this approach, the data structures must first be defined, and then the program elements are structured based upon the data structure itself. This then is an attempt to clearly and precisely explain the implementation of the objects in the solution space and then allow their specific structure to become visible to the essential functional elements that furnish the operations on the objects. In general, data structure design defines a set of "mapping" procedures that use information (data) structure as a guide [Ref. 3: p. 141]. This approach recognizes the necessity for the design of the program to reflect the structure of the problem.

3. Parnas Decomposition Criterion

The Parnas decomposition criterion methodology is based upon the idea that as a system is decomposed, each module in the system hides a design decision from the other

modules. This approach is implemented through an initial identification of difficult design decisions or design decisions that are likely to change over time. Each program module is then designed to hide such a decision from the others. This results in the capture of design structures in the software at the level at which the design decision is made. If modification of the software system should become necessary, ability to minimize the effects of the modification should be readily available. This criterion also supports the idea that to achieve an efficient system implementation, the assumption that a module is one or more subroutines must be abandoned in favor of allowing subroutines and programs to be assembled collections of code from various modules [Ref. 7: p. 225].

4. Object Oriented Design

Object oriented design is a relatively new approach to software design that has developed as a result of the works of various people in the discipline [Ref. 8: p. 38]. This methodology allows the mapping of solutions directly to the designer's view of the problem. The object oriented design approach first clearly and concisely defines the problem. An informal strategy is then developed to provide initial direction toward the solution of the problem. Finally, the strategy is formalized. During this step, identification of the abstract objects at given levels in the

system takes place. The appropriate operations on these objects are then defined. Interfaces are established and operations are implemented. The last step is to develop a module that hides the implementation.

This methodology provides a meaningful strategy for decomposing a system into modules, where design decisions are localized to complement the real world view. This approach also provides a consistent notation for choosing the objects and operations that form the design. The object oriented design approach provides an enforceable structure which should ease some of the complexities involved in software system design.

D. THE USE OF PROGRAMMING LANGUAGES AS SOFTWARE DEVELOPMENT TOOLS

Software system design methodologies are not sufficiently capable of producing computer solutions on their own. These approaches require the assistance of software tools, particularly through the use of programming languages, to express and execute design. In order to discuss the evolution of programming languages into efficient software system design tools, a language generation outline of the most popular programming languages and some language features are provided [Ref. 9]:

First Generation Languages (1954-1958)

FORTRAN I
ALGOL 58
Flowmatic
IPL V

Second Generation Languages (1959-1961)

FORTRAN II	subroutines, separate compilation
ALGOL 60	block structure, data types
COBOL	data description, file handling
LISP	list processing, pointers

Third Generation Languages (1962-1970)

PL/1	FORTRAN + ALGOL + COBOL
ALGOL 68	rigorous successor to ALGOL 60
Pascal	simple successor to ALGOL 60
SIMULA	classes, data abstraction

The Generation Gap (1970-1980)

Many different languages, but none endured.

As can be readily seen from the outline above, the more commonly utilized high order languages, FORTRAN and COBOL, came into existence in the early history of computer science, before the advent of the "software crisis." Accordingly, these high order languages were not founded on modern software design principles and, as a result, these languages had to be modified by use of preprocessors (S-FORTRAN) and extensions (FORTRAN-77) to bring them into compliance with current software design methodologies. Needless to say, these languages were formulated prior to the recognition and acceptance of the fact that large, modern software systems are far too complex to efficiently manage.

These high order languages continue to fulfill needs of their individual problem domains; however, since their creation, the large embedded computer systems domain has arrived. None of these high order languages was designed to cope with the inherent complexity associated with embedded systems.

A discussion of the basic structure of these high order languages will demonstrate some of their intensive problems [Ref. 8: p. 34]. FORTRAN and COBOL were both designed with flat structures, primarily made up of global data and one level of subprograms. The inherent danger associated with this type of structure is that an error introduced in any segment of a program can result in catastrophic ripple effect across the entire system due to the global data structure. Modifications to large systems utilizing these high order languages generally result in the disintegration of the original software system design structure. Maintenance on programs written in these languages often produces large amounts of cross coupling among program units, resulting in a lessening of system reliability and solution clarity.

Most of the second and third generation languages became capable of providing a larger nested structure for more complex algorithms. However, there was little or no improvement in the ability for describing data structures. The basic structure of these languages was very similar to

that of the first generation languages with the major difference being the existence of subprograms within subprograms. Unfortunately, these languages were plagued with the same problems inherent to the first generation. Some languages in this generation such as SIMULA, did demonstrate the ability to provide greater data structuring. However, these languages failed to gain any sizeable credibility.

Assembly languages are presently the most commonly used languages for embedded systems. Assembly languages exhibit no inherent structure. As a result, assembly languages provide great flexibility in developing systems and assembly languages can be written in structured assembly code. However, once a system becomes fairly large, the mere nature of the language tends to confuse the organization.

The evaluation of fourth generation languages, such as Ada, are already demonstrating tremendous potential in the alleviation of the problems associated with the description of data structures. These languages are able to control system complexity through physically concealing unessential details at each system level. Their basic structure supports the localizing of design decisions, and maintains the structure of the original design as modifications are made.

III. SURVEY OF CURRENTLY DEVELOPING PROGRAM DESIGN LANGUAGES

During the past several years, industry has seen an explosion in the cost of software production coupled with a decline in the quality and reliability of the results. A realization that structured programming, top down design, and other changes in techniques can help has alerted the field to the importance of applying advanced design and programming methods to software production [Ref. 1: p. 5]. One of the most promising software system design tools to emerge from the appreciation of this problem has been the development of the program design language (PDL) concept. This chapter will define the program design language concept through a discussion of its functions and attributes, and conclude with a description of currently developing PDLs as outlined in Reference 10.

A. PROGRAM DESIGN LANGUAGE CONCEPTS

The term program design language is used synonymously with other recognized software engineering terminologies such as pseudocode, structured English, and metacode. However, for purposes of discussion, the acronym PDL will be utilized exclusively throughout this chapter and the remainder of the thesis. Conceptually, a PDL is a very high order programming language designed to relate the logic of a program module in

an understandable and readable format at any given level of detail. PDLs were originally designed for a top down approach to software system design development. It is considered a "pidgin" language in that it uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e., a structured programming language) [Ref. 3: p. 253].

On the surface, PDLs appear to be very similar to the existing third generation programming languages developed in the 1960's. However, a major dissimilarity exists in the fact that PDLs utilize English as a narrative text embedded expressly inside PDL statements. PDLs combine this narrative text with a formal procedural format that forces upon its users a programming language-like syntax which enables automated tools to assist in the development of detailed design. Presently, the combination of the narrative text with the formal procedures makes compilation of PDLs impossible. However, this set of automated tools known as PDL processors, make it possible to design operational indices, format text, produce cross reference tables and nesting maps, check validity of the syntax, and perform several other functions.

The input to the PDL processor is comprised of control information and designs for procedures known as segments. These segments are utilized to describe the algorithms used in performing the mandatory steps contained in a program

module. They make up the interface specifications between modules and are used to define the functions performed by a given module. Since PDLs utilize module structure in their architectural design, each segment contains only a small portion of the overall system logic found within the module. This use of PDL segments results in the creation of algorithms that are more precise, easily understood, and more rapidly modified supporting the statement that, "The purpose of a design is to communicate the designer's idea to other people--not to a computer" [Ref. 11: p. 271].

The output from the PDL processor is in the form of a working design document. This output has been recognized as an extremely effective replacement for conventional flowcharts. There are several apparent reasons why PDLs are effective in accomplishing this:

1. They are machine-processable, using the text editing facilities available in the software development environment.

2. They can be easily read, so that a group of designers can easily review the PDL of a given designer to determine the quality of the design (structured walk through).

3. They are read in a top down manner and provide a more accurate reflection of the program structure than do flowcharts at a larger stage in software development

Through the example of simple sorting algorithm, Figures 3-1, 3-2, and 3-3 [Ref. 19] clearly demonstrate the overwhelming clarity inherent in PDL output documentation

vice that of conventional flowchart or a third generation programming language (PL/I) presentation.

```
SORT (TABLE, SIZE OF TABLE)
    IF SIZE OF TABLE > 1
        DO UNTIL NO ITEMS WERE INTERCHANGED
            DO FOR EACH PAIR OF ITEMS IN TABLE (1-2, 2-3,
                       3-4, ETC.)
                IF FIRST ITEM OF PAIR > SECOND ITEM OF
                       PAIR
                    INTERCHANGE THE TWO ITEMS
                ENDIF
            ENDDO
        ENDDO
    ENDIF
```

Figure 3-1. PDL Design of Simple Sorting Algorithm

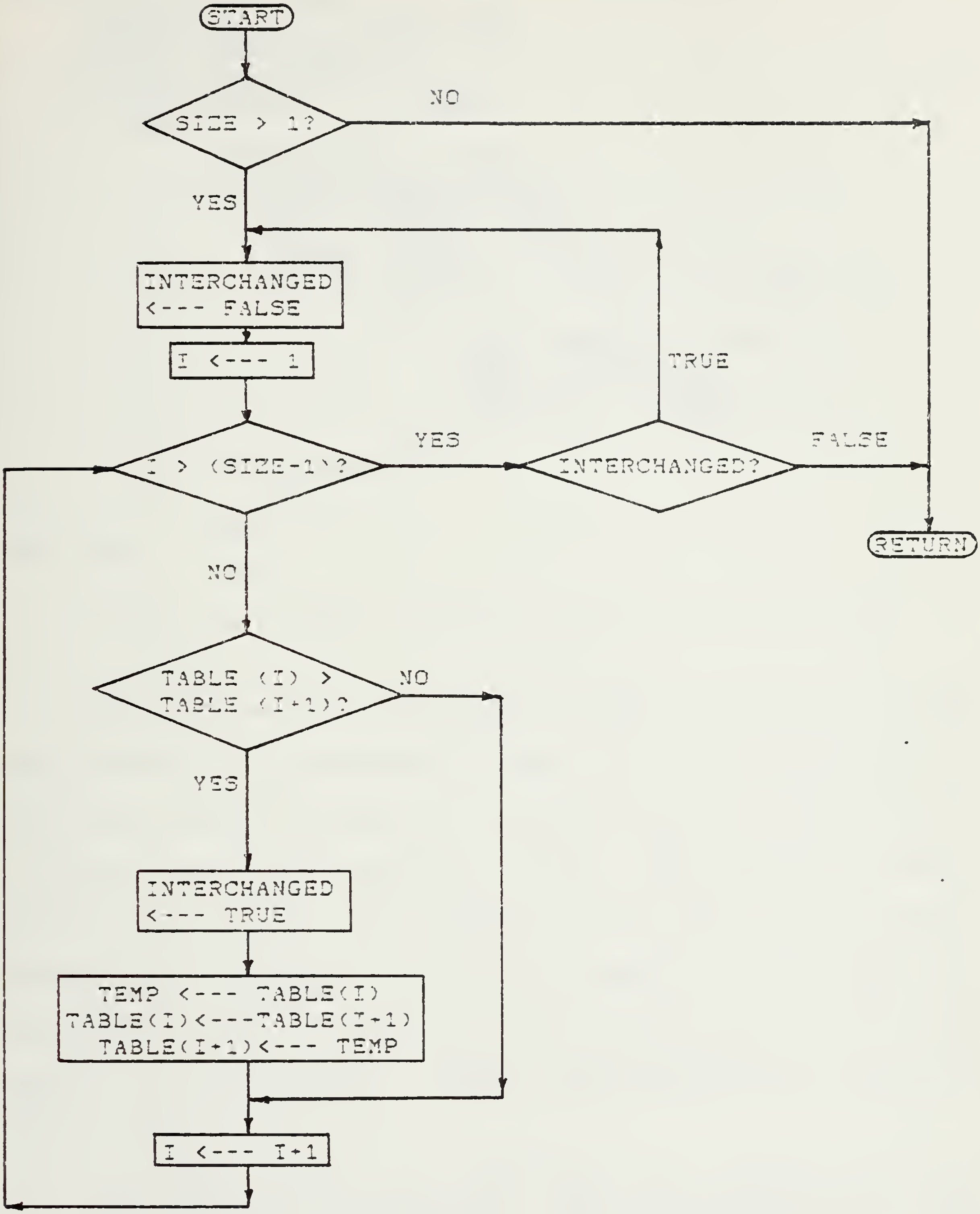

Figure 3-2. Flowchart for Sorting Algorithm of Figure 3-1

```
SORT:
    PROCEDURE (TABLE):
    DECLARE TABLE (*) FIXED BIN:
    DECLARE INTERCHANGED BIT (1):
    DECLARE TEMP FIXED BIN:
    IF DIM (TABLE, 1) > 1 THEN
        DO:
            INTERCHANGED = '1'B;
            DO WHILE (INTERCHANGED);
                INTERCHANGED = '0'B;
                DO I = LBOUND (TABLE, 1) TO
                HBOUND (TABLE, 1) -1;
                    IF TABLE (I) > TABLE (I+1) THEN
                        DO:
                            INTERCHANGED = '1'B;
                            TEMP = TABLE (I);
                            TABLE (I) = TABLE (I+1);
                            TABLE (I+1) = TEMP;
                        END;
                    END;
                END;
            END;
END SORT;
```

Figure 3-3. PL/I Procedure for Sorting Algorithm

These figures lend themselves to demonstrating one of the most important attributes of PDLs; the ability to quickly develop a coarse profile of a problem solution that is both readable and understandable by all. This aids in design modification since individuals at all levels in the system design development phases are capable of quickly and accurately identifying errors and potential problems and ensuring correctness. Other characteristics that PDLs should comply with are:

1. A fixed syntax of KEYWORDS that provide for all structured constructs, data declarations, and modularity characteristics.

2. A free syntax of a natural language that describes processing features.

3. Data declaration facilities that should include both simple (scalar and array) and complex (linked list or hierarchical) data structures.

4. Subprogram definition and calling techniques that support various modes of interface description.

5. A PDL should be programming-language-independent. A design described with a PDL should be translatable to assembly language, FORTRAN or PASCAL [Ref. 3: p. 253].

PDLs in themselves are not a panacea for the ills that affect software system design. However, if the PDL concept is utilized effectively, the goals and principles of software engineering can be achieved quite successfully.

Another software design concept which is gaining greater acceptance throughout the discipline is that of a System Design Language (SDL). Whereas PDLs provide a detailed description of a program module, SDLs can be viewed as a module interface language for a format architectural description of a system. The SDL concept is a logical outgrowth of the PDL concept. An SDL ideally will identify those system components needed to be constructed and what interfaces each component provides and requires. A PDL, on the other hand, identifies how each component is to be constructed. Together, these concepts form the "blueprint" for actual software system implementation. Although these concepts are closely related, a formal discussion of System Design Languages is beyond the scope of this thesis.

B. SURVEY OF CONTEMPORARY PDLS

Several software vendors have been developing PDLs to assist them in the creation of software systems. This research and development effort has been stimulated to a large extent by the DOD initiative concerning the development of Ada for use with embedded computer systems. Many of the PDLs now being developed are in fact Ada-based design methodologies.

Four of the most promising PDLs currently under development are being built by Harris Corporation, TRW, IBM and Norden Systems. Each of these PDLs has a distinct syntax and each supports a dimunitive variation in design methodology. A brief description of each PDL follows. Additionally, a matrix demonstrating the Ada language features supported by each vendor's PDL is included as Appendix C.

1. Harris PDL

The PDL developed by Harris exceeds the confines of conventional PDLs by including guidelines for the software development process. Harris redefines the term PDL to mean 'Process Description Language' to reflect the extended application for the language. This PDL utilizes two constructs for clarity enhancement, a 'call' keyword prior to

subprogram calls and an 'engage' keyword prior to task calls. The Harris PDL also embodies six keywords for file input/output. These are 'open', 'write', 'read', 'close', 'delete', and 'create'. Harris also permits the usage of structured English statements in the form: VERB NOUN/OBJECT (OPTIONAL MODIFIERS).

For the program development, the Harris PDL utilizes four approaches. The first approach is top-down partitioning which is used to break down the system into layers of mutually exclusive subsystems, which, when taken as a whole, totally encompass the original design. The second approach is known as progressive elaboration. This approach is used in conjunction with the top-down partitioning approach so that as the system is being broken down layer by layer into modules, detail is progressively added to the data and process structures. Horizontal compilation is the next approach. In horizontal compilation, as top-down partitioning occurs, each layer can be computed to test for consistent and complete partitioning as well as correct syntax. Vertical verification is the fourth approach. This method is used to insure that nothing has been left out or added in succeeding layers of partitioning. The Harris PDL effectively lends itself to the achievement and support of the software engineering goals and principles.

2. TRW PDL

The TRW Corporation is developing a PDL based primarily on the Ada programming languages for use by the Department of Defense. Since the issue of the utilization of Ada as a PDL will be fully discussed in a subsequent chapter, the description of the TRW PDL will be of limited scope.

The TRW PDL supports the basic constructs of the Ada programming language, i.e., packages, compilation units, tasking, generics, typing of data and data hiding. However, some Ada language features are not supported by this PDL. Simple statements such as null, procedure call, abort, and assignment lack proper supports.

The most significant feature of the TRW PDL is that it permits the insertion of narrative text into statements in lieu of Ada comments. Although this is viewed by Ada PDL supporters as a potential problem, it indicates that TRW is attempting to develop a PDL with a certain degree of programming language selection flexibility.

3. IBM PDL

IBM has been working on program design languages for several years. The methodology for the development of its PDL is focused on the following design language requirements:

1. Enforced recording of both interfaces and behavior specifications as part of the design of the software.

2. Imposition of structure while allowing for free-form expression of specification ideas.

3. Data declaration facilities to allow definition of both individual scalar values and data groups.

4. Definition of user defined data types.

5. Definition and use of procedure and functions to provide modularity.

6. Concurrent assignment notations that one can express in a design the situation of several inputs producing several outputs in an unspecified sequence.

7. Encapsulation and information hiding.

8. Formal commentary with specified format and scope.

9. Support for stepwise refinement of the design.

Currently utilized third generation programming languages possess many, but not all, of the attributes listed above. The growing popularity in utilizing Ada as a base for a design language is due to the fact that these forementioned attributes are directly embodied in the Ada language. IBM is developing, in parallel with its generic PDL, an Ada PDL which encompasses these attributes with other Ada concepts.

4. Norden PDL

Norden Systems has also been developing PDLs for some time. Their PDL is a non-compilable PDL similar to the Caine, Farber, and Gordon PDL discussed in Reference 11. This PDL l backs the more powerful language features such as strong typing, loop exits, and tasking and, as a result, Norden has opted to develop an Ada oriented PDL. This new PDL, NPDL/Ada, utilizes an Ada syntax, embodies the major Ada language features yet retains the expressive freedom of the

English language text while embedded in the more rigid Ada control structures.

The software vendors who are creating PDLs, are quickly recognizing the inherent value of utilizing the Ada programming language as a basis for the development of PDLs. The concepts and features that make the Ada programming language so conducive to utilization as a PDL will be discussed in the next chapter.

IV. EXAMINATION OF ADA CONCEPTS

A. BACKGROUND

The explosive growth in the cost of developing and maintaining complex software systems has fostered the advancement of a large number of techniques and theories developed in the area of software design and development. These theories and techniques include structured programming, top-down design and implementation, structured analysis and design, modularization, and programming teams and walkthroughs. The central aim of these theories and techniques has been to attempt intellectual control of a software system design via systematic decomposition and abstraction of the software problem into component modules and subsequent composition of those modules into the system [Ref. 11: p. 220]. Only with an intellectual control and working understanding of large, complex software systems can the development and maintenance costs of those systems be kept in check.

Recognizing that development of the Ada language provided for a unique opportunity to discard the inefficiencies of older generation languages while creating an entirely new method with which to allow intellectual control over complex system software, the HOLWG established three guiding principles or goals early in the Ada development process. These goals are:

- recognition of the importance of program reliability and maintainability;
- concern for programming as a human activity; and
- efficiency.

The finalized STEELMAN document reflected the desirability of these three goals by mandating that the final Ada language support the following language features [Ref. 8: p. 18]:

- structured constructs;
- strong typing;
- relative and absolute precision specification;
- information hiding and data abstraction;
- concurrent processing;
- exception handling;
- generic definition; and
- machine dependent facilities.

B. LANGUAGE OVERVIEW

Of the three goals envisioned ~~by the HOLWG~~, the first and most important was considered to be that of reliability and maintainability, as together these make up the largest cost areas in a software system's life cycle. In an attempt to maximize reliability and maintainability of Ada software systems, Ada was designed to be, and is, a design language. As such, it focuses primary attention upon the interconnection of the interface characteristics of the components

within a system rather than upon the components themselves. Somewhat like the manner in which a blueprint describes the way things fit together without extensive detail as to what those "things" are, Ada emphasizes the interconnection between module interfaces over the structure within those modules. Further, it is the interface characteristics which actually define the components which are used in the design because the interface characteristics are all that the user of the component needs to use the component and all that the designer needs to design the component. This approach to language design specifically supports modularization, information hiding and abstraction as the user need not see the contents (how) within each module, but rather he need only see the interface (what) and interconnection of modules. This approach is also multi-tiered, as within each module there exists a system of interconnectivity between interfaces of lower level modules, down to the level where further decomposition becomes inappropriate. An Ada software system, then, can be viewed in its entirety as a single module with the components of that module being the interconnections of interfaces of lower level modules, and so on down to the lowest level modules in the system.

C. OBJECT ORIENTED DESIGN

A methodology with which to approach the design of a software system where that system is intended to solve real

world problems has been advanced by Grady Booch [Ref. 8: p. 40]. His methodology, called object-oriented design, begins with the recognition that there is a problem space wherein real world problems are begging of a solution, and there is a solution space wherein computer software and hardware combine to accept real world problems, process those problems toward solution, and inject those solutions back into the real world. In any programming language the programmer translates (abstracts) real world problems from the problem space (real world) to the solution space (software). The software/hardware system operates on these abstractions toward a solution by way of an abstract problem-solving technique (software algorithms), and the solution is then converted back to the real world by way of computer output. The primary problem with computer languages prior to Ada is that a considerable distance exists between the problem space and the solution space, resulting in the need to expend considerable effort in both converting (abstracting) to and from the solution space and operating efficiently within the solution space arena.

The closer the solution space maps to our concept of the problem space, the better the goals of modifiability, efficiency, reliability and understandability can be achieved. Most of the languages developed prior to Ada are primarily imperative; that is, they provide a rich set of constructs

for implementing operations within the solution space but are generally weak when it comes to abstracting real-world objects into that solution space. Additionally, these languages require that the real-world problem space, which is both multi-dimensional in description and highly parallel in effect, be mapped into a solution space that has a relatively flat topology as well as a high dependence on sequential processing for solution attainment.

The Ada language, when viewed through the window of Booch's object-oriented design methodology, allows us to minimize the distance between the problem space and the solution space by emphasizing the fact that object-oriented design is not a purely functional design technique. Rather, it recognizes the importance of treating software objects as actors, each with its own set of applicable operations.

The three steps to object-oriented design, along with a brief description of each, follows.

1. Define the Problem

At this stage we remain entirely within the problem space, and attempt to gain an understanding of the structure of the problem space at hand. This step will be iterative, working from the general to the specific, and such tools as SADT and data flow diagrams are wholly appropriate in defining the problem.

2. Develop an Informal Strategy

Once an understanding of the problem space is gained, an informal strategy as to how to arrive at a solution is in order, where that strategy parallels our view of the real world. This strategy is best kept within the realm of natural English descriptions and in terms of concepts existing in the problem space. In this way intuitive feel as to how to solve the problem is not yet lost among the complexities of abstraction into the solution space.

3. Formalize the Strategy

In this step we finally enter the realm of the solution space and incur the need for abstracting from the problem space to the solution space. From the informal strategy already developed, we first extract the nouns which represent objects in the problem space and then the qualifying adjectives which represent attributes in the problem space of those nouns they qualify. Nouns in the English language can be common nouns (such as table or chair), mass nouns (such as water or fuel), or nouns of direct reference (which refer to a specific object in the problem space). Adjectives identify the attributes or constraints of these nouns, and when such adjectives as "concurrent" and "asynchronous" are used in the informal strategy, the parallel nature of the problem space is revealed.

Having extracted the nouns and adjectives from the informal strategy as objects and qualifiers to those objects, respectively, we must then extract the verb phrases occurring in the strategy. In doing so, we identify the real world operations being performed on those objects in the strategy and further associate each operation with a particular object in the problem space against which each operation acts.

Perhaps the most important step in formalizing the strategy involves establishing the relationships among the objects already defined. By this it is meant that the visible interfaces to each object are identified and formally described using Ada as the design language. By identifying the object interfaces and their relationships (interconnections), a contract is formed between the user of an object and the object itself, and this contract explicitly defines the operations which may be performed on the object by the user. The beauty of Ada is that it not only permits us to easily describe such a contract but also enforces the contract by preventing us from violating our logical abstraction.

The contract having been made as to the permissable operations useable against any particular object, we can then implement those operations in the Ada language. This results in operations which are executable, and further allows the development of a design for solution to the problem which is

also executable. The implementation of operations and the design for solution to the problem will naturally reveal lower level objects and operations needed to support the present level of solution implementation. These lower level objects and operations can in turn be addressed, leading to further iterative decomposition until the point is reached where further decomposition will not aid in system understandability.

D. ADA LANGUAGE TOOLS

The fundamental building blocks of the Ada language are program units, and every Ada program is made up of these program units [Ref. 8: p. 47]. Each program unit is made up of two distinct parts: a specification, which contains those entities visible to other program units and which thus defines the external characteristics (interface) of the program unit, and a body, which contains the implementation details of the program unit where those details are not visible to other program units. The specification part and the body of any program unit can be separately compiled, which greatly enhances the management of designing an Ada software system. This is because at any level of software system design, it is only necessary to write the specification parts of the program units used at that level which can then be compiled resulting in the creation of an enforceable design structure to the problem solution. An added benefit

to the process of making distinct the specification part and body of Ada program units is that it encourages both the construction of systems from separately built parts and the construction and use of libraries of generally useable component modules.

Ada program units are categorized into three distinct areas: subprograms, tasks and packages. Each of these categories is explained below.

1. Subprograms

Subprograms are the basic units for expressing algorithms and provide the means for naming definable functions. They have the characteristic of being sequential in execution and can range in scope from being the main program down to being a lowest level module. The subprogram specification defines the interface, or calling convention, between the subprogram and the outside world while the subprogram body encapsulates the algorithm for which the subprogram exists. The applications for Ada subprograms include main program units (the highest level of an Ada software system), definition of functional control (where the functions determined at one level of design are implemented via subprogram at the next lower level), and definition of type operations for abstract data (where user-defined or abstract data types can be linked with unique operations regarding those data types).

Subprograms have two basic forms: procedures and functions. A procedure provides the series of actions which are defined in its body whenever that procedure is invoked, and it may have parameters to pass information either to itself or back to the invoking unit. A function has the primary purpose of returning a calculated value where that value is computed within the function and returned to the program unit which called the function.

2. Tasks

Tasks are the program units which define operations or procedures which execute in parallel with other tasks. Where most existing high-order languages provide little or no support for the parallel execution of program operations or procedures, Ada specifically accomplishes such parallel execution through the use of tasks. Since the real-world problem space operates in a highly parallel fashion (more than one event occurring at a time), the task program unit serves to greatly reduce the distance between the problem space and the solution space by eliminating the need to convert real-world parallel events into the serial abstraction demanded by other high-order languages. Physically, tasks may execute on multicomputer systems, multiprocessor systems, or with interleaved execution on a single processor. As such, tasks can be seen as individual sequential processes, where each process interacts with other processes

through a sophisticated means of communication and synchronization among individual tasks. The term "rendezvous" applies to the place and time at which two individual tasks interact, and it is this interaction among tasks that allows, for instance, one task to detect and report the inaction or improper action of another task. Such an approach enhances communications reliabiliy and error detection within the software system.

Like the subprogram and package, the task is divided into a task specification, which defines the interface between the task and other program units, and the task body, which consists of the task's executable part.

3. Packages

Packages are the units used for encapsulating collections of logically related data, objects and data types. The package specification defines the interface to the package and thus specifies which parts of the package may be used and, furthermore, how they may be used. The package specification may be further divided into a visible and a private part, where the visible part declares the package resources which may be used outside the package, and the private part which, while textually available to the package user, cannot be referenced outside the package. The package body is specifically ngt accessable outside the package, and contains

the necessary sequence of statements relating to the package purpose.

Packages are extremely versatile as to their possible application, and the logical grouping of objects and data types places the definition of those objects and data types in one location. This application greatly enhances maintainability. For instance, if changes become necessary within a logical grouping, only the single package need be changed thus ensuring consistency throughout the system for any program unit calling that package.

With packages it is also possible to group logically related program units, namely, subprograms, tasks, and even other packages. The advantage here is that the algorithms or contents of those program units within a package can be changed (for, e.g., reasons of efficiency) without affecting the program units which call the package. Packages also allow the user to uniquely define an abstract data type and then encapsulate it in such a way as to enforce the abstraction through the Ada language. Thus where a set of data types are unique to a specific application, those data types and their application can be placed within a package, and that package will then disallow improper implementation of those types.

E. DATA TYPING

The objects within the Ada language equate to the nouns we use in everyday English. Each object in Ada has a set of properties which denotes the kinds of values that the object can carry and the operations which we can apply to that object. This set of properties is called the object's "type" in Ada. The types applicable to any object in Ada must be specifically declared within the software system as these types do not exist implicitly within Ada as they do in other high-order languages such as FORTRAN. Data typing within Ada has the effect that objects of a given type may take on only those values that are appropriate to the type and, in addition, the only operations that may be applied to an object are those that are specifically defined for its type. Because of this, Ada is recognized as a strongly typed language. Strong typing within Ada provides a mechanism for imposing structure on the data manipulated within an Ada program and, in addition, directly supports several of Ada's recognized design needs, including maintainability, readability, reliability and reduction of complexity.

There are four intrinsic data types within the Ada language--scalar, composite, access and private types. Scalar types include both numeric types (including integer, fixed point and floating point) as well as enumeration types (which allow the programmer to assign ordered sets of

specific enumeration literals to be used as values in the program). Composite types include array types, which allow the collection of similar or homogeneous objects in an indexed form, and record types, which allow the collection of potentially different or heterogeneous objects within the record. Access types are designed to handle those objects which are subject to dynamic change over time and even during program execution, such as buffer space within a message-passing system or geneaological records in a data base.

The last category of data typing, the private type, is the most inventive of Ada's data typing tools. Declared within the package specification, the private typing of objects serves to hide within the body of the package both the structure of the data used to define the type and the algorithms which implement the operations on that type. Only the names of the private types within a package are visible to the users of that package. The primary benefit of private types is that they support directly the principle of information hiding wherein the details of an implementation are suppressed in order to allow focus on the abstraction of lower level modules.

F. GENERIC PROGRAM UNITS

One potential disadvantage to Ada's strong typing rules is that multiple forms of packages and subprograms may have to be designed in order to process objects of different

types, even though the algorithms within those packages or subprograms are identical. This is because Ada's strong typing rules require us to specify the type of every object at compilation time and, if the object's type does not "fit" the package or subprogram specification, it will be denied entry to that package or subprogram. To deal with this problem, Ada has as one of its tools the generic program unit. The generic program unit serves as a "prefix" to what would otherwise be a non-generic program unit and it allows access to the program unit for all generic parameters named in the generic unit. The benefit of the generic program unit is that a general purpose program can be written just once but used many times and by different program units.

G. INPUT/OUTPUT

Embedded computer systems have a requirement that the computer communicate with I/O devices which are oftimes unique to that system which has usually required that software coding be employed to specifically match the computer with its I/O devices. This coding is always tedious and costly and almost never portable to other systems. On the other hand, where only one type of formatted I/O is used for a particular application, most implementations of existing high-order languages will bind a huge routine library unit that will handle virtually any kind of formatted I/O, whether we use those features or not. With Ada there

exists the ability to build I/O routines for communicating with unique devices and, while the routines themselves may be tied to the devices they serve by virtue of device uniqueness, the parts making up the routines as well as those servicing the routines, are portable. Additionally, Ada allows for the utilization of redefined units for I/O of common data types which can be selected as needed without the need for adding any new language constructs.

H. DOCUMENTATION

A significant advantage to using Ada in a software system design is that the means of documenting the structure of the system ultimately becomes the same means with which the system is implemented. In fact, where Ada is used as both the design and implementation language within a system, the maintenance of the design documentation becomes automatic since such maintenance is an integral part of the implementation process. Thus, whereas with other design processes the design is documented in a form wholly different from the implementation language and thus requires a two part effort in design maintenance or change, with Ada every change in the Ada implementation will, in principle, update the documentation commensurate with that change. Additionally, since Ada is a highly structured language, it is easy to maintain the original structure of the system while modifying the underlying pieces.

I. LIFE CYCLE ISSUES

Though only a very small part of the virtues and tools inherent in the Ada language have been touched upon here, it can be seen that the underlying philosophy behind Ada as well as the implementation tools available with Ada combine to form a software language system that will greatly enhance the ability to manage Ada software systems over their life cycles. Traditionally, software developers have taken a restricted view of the life cycle process and have treated each phase of a system's life cycle as an independent part. This approach has lead to numerous problems, including configuration control nightmares and sets of software modules that would not function together. In the end, the developers would complete the systems they started, although probably not on time and not within budget.

Ada will not solve the software crisis by any stretch of the imagination. It will, however, avert the transition of that crisis into a software catastrophe if it is expeditiously and judiciously applied to prospective and future software development projects. It will do this by allowing all levels of management and implementation to maintain control over the systems they are tasked to develop, where that control today is in large part absent and sorely needed.

V. UTILIZATION OF ADA AS A PROGRAM DESIGN LANGUAGE

A. PRESENT UNDERLYING PROBLEMS

The discussion of different private endeavors to design a usable PDL presented in Chapter III points to a revealing and somewhat distressing fact: there is apparently no consensus as to what an Ada PDL should consist of or how it should be used. For example, where the Harris PDL supports all features of the Ada language and in fact incorporates two additional non-Ada constructs to add clarity, the IBM PDL is a strict subset to the formal Ada language, and the TRW and Norden PDLs allow annotations to the Ada language for use as a PDL. This lack of consensus is further exemplified by the fact that at least one vendor does not consider the acronym "PDL" as meaning program design language (in the case of Harris, the acronym means "process description language").

One is forced to ask under these circumstances whether present attempts to design and construct an effective and usable Ada PDL are approaching that end in the most effective manner. Of course the answer to this question is that the individual efforts, however different they may be from one another, each have attributes which contribute positively to the desired end goal of creating an Ada PDL. It is not yet known which, if any, of the mentioned vendors' designs will

be adopted by DOD, or whether a combination of design attributes from among several vendor designs will be adopted. What can be said about the different designs mentioned is that there is a common thread of enthusiasm and support among vendors that they key elements of the Ada language directly support the process of program design.

But enthusiasm alone does not beget a usable end product, particularly in the area of software development. The management of a software development project is perhaps no different than the management of other large engineering projects, except that the end product is certainly less tangible than, say, a bridge or a ship. An ideal situation, and one that would greatly simplify the management of software engineering, would be the existence of an automatic program generation system as the ultimate PDL, but of course the discipline of software engineering has not yet progressed to that point. What is needed, then, is a program design language that will maximize the manageability of any software system development where the primary tool used in that development is the PDL adopted.

B. AN EXAMPLE OF PDL/ADA IN PROGRAM DEVELOPMENT

Perhaps one of the most revealing studies into the problems inherent in the design and implementation of a PDL was a study conducted by General Electric and the University of Maryland under contract with the Office of Naval Research

(ONR) in 1982 [Ref. 12]. In this study an attempt was made to systematically measure some of the major problems associated with a software development project through the vehicle of actually having a program design team develop a mock project utilizing Ada as a PDL. Following an intensive, month-long training program into concepts and operations of the Ada language, a three-member program design team set out to design a portion of a working ground support system for communications satellites. The system was already in existence in the programming language FORTRAN, and one intent of the study was to compare both the time and effort in development as well as the functionality of the end-product program with the existing FORTRAN program.

The program development process was divided into two distinct phases. The first phase was the design phase and it involved creating a brief description of each known component in the system. This design phase was intended to be written in compilable Ada, vice flowcharts or other means, and as such this phase encouraged the use of the entire Ada language inventory as a PDL. The second phase involved writing a more precise design, including specific algorithms, complete interface specifications, the definition of all data types and the declaration of all data objects. Following the second phase each design component was coded in Ada resulting in an executable software end product.

The use of two distinct design phases was not initially set out as a requirement in the design of this program. In fact, at first the design team was given a free hand as to their design style, and they began the program design process with only one design phase intended. It was recognized almost immediately, however, that when the primary emphasis was to design using compilable Ada, the resulting design evolved as increasingly detailed threads of functionality rather than as complete descriptions of the system at each level. That is, each team member tended to follow one function through the various design levels, filling in greater detail at each lower level for that particular function, rather than providing a complete description of the system at each level before attempting further refinement at lower levels. As a result of this tendency at vertical program development vice horizontal development the two-phased design approach was imposed, and it was with this approach that the problem ran to completion. An additional problem was recognized after the final program design was completed. While the final design was judged to be a good and workable design, it was characterized as being a highly functional one and one very similar to the original FORTRAN based system, even though the design team had no direct access to the original FORTRAN design. While this may not at first seem to indicate a fundamental problem with the final

design, it does raise two issues regarding that design. First, the design indicated a failure to take full advantage of the design power inherent in the Ada language, and second the question is raised as to whether an alternative design approach was not considered, such as Grady Booch's object oriented design methodology [Ref. 3]. Although examples of data abstraction and encapsulation were presented in the Ada training course, the emphasis was placed on language features that support those ideas rather than the ideas themselves during training. As a result of the apparent failure to consider alternative design approaches in this problem, the study concluded that a more expansive training program would be advisable in future programs of this nature. Such a training program would specifically address alternative design approaches since a choice of alternatives impacts the initial design decisions and perhaps even the requirements analysis phase.

C. MANAGEMENT ISSUES

From the discussion of underlying problems and the example of an Ada design project presented above, it becomes apparent that some fundamental management issues must be addressed before Ada (or any other new programming language) can be specifically implemented as a program design language. Some, though not all, of those issues follow.

1. The Need for Education

The General Electric/University of Maryland study reveals one of the most important management issues which must be specifically and comprehensively addressed before Ada can be adequately designed into and utilized as a PDL--the need to re-educate designers in the Ada language. The primary reason for this need is that virtually all of today's programmers and program analysts were trained to understand the conventional process oriented-design methodology as the only means with which to design or program computer software. While process-oriented design is not in and of itself "bad" design methodology (it has been the primary means of software design since the onset of the computer age), it does have limitations in its application to program design primarily because it focuses attention on "how" processing is taking place rather than on "what" is being processed. The usual result of this focus has been that program designers have tended to devote too much energy toward the intricacies of the software itself while losing sight of the overall purpose for which the software was created. Such a focus usually results in software that is so overly complex and interdependent as to require the injection of patchwork languages just to get the software system to work, with a resulting product that is all but unmanageable. This is the single most identifiable cause of the present software crisis.

If the tools available in Ada are to be effectively utilized in the design of a PDL, then those responsible for that design must be highly versed in the object-oriented design strategy and methodology. The shift in one's thinking away from process-oriented design and toward object-oriented design requires more than a shift in method or technique; it requires a fundamental shift in software design philosophy. The required shift in thinking is so fundamental, in fact, that some have argued that the untrained might be easier to educate in the object-oriented design methodology than those already trained in process-oriented design [Ref. 13]. There is a clear need, then, to ensure adequate education for the designers and users of the Ada PDL, as well as for the users of the language itself in applications programs.

2. The Need for Standardization

Clearly the present software crisis will only be replaced by a new software crisis if adequate controls as to standardization are not enforced in the Ada environment. Each of the four vendors mentioned in the SofTech study had a unique approach as to what a PDL should do, and how to design a PDL in the Ada language [Ref. 10: p. 3-18]. Somewhat distressing is the fact that two of the vendors introduced annotations to the Ada language in an attempt to create their respective versions of an Ada PDL, which suggests the possibility that a whole new group of branch languages might

eventually evolve from the present Ada language. One of the basic precepts to the creation of the Ada language was to discourage such branching in an effort to maintain manageability and maintainability of Ada software.

In designing an Ada PDL or any other Ada based software, it is imperative that we not lose sight of the original intent of the language as put forth by the HOLWG--that or maintaining standardization of the language's application. Some diversion from the original language may be necessary in order that a workable end product be developed, but that diversion should at all costs be kept to a minimum.

3. The Need for Horizontal Vice Vertical Design

Ada has been described as an ideal tool in the design of both program design languages (PDLs) and system design languages (SDLs) [Ref. 14]. Where a PDL describes how each component in the software system is to be constructed (including control flow), an SDL description shows what components need to be constructed and what interfaces each component provides and requires. The two features of Ada which distinguish it from other languages and which make it an ideal program and system design tool are that it is an object-oriented language and that all packages and subprograms are broken down into specifications and bodies, each of which are separately compilable.

The beauties of an object-oriented design structure have already been touched upon--they discourage designers from getting "lost" in the intricacies of the solution space while forgetting the purpose for which the software system was created in the first place. The primary beauty of separate compilability between specifications and bodies is that it allows the emphasis to be placed on horizontal development within a system or component prior to the need for vertical development within that system or component. By horizontal development, it is meant that all elements or components existing within a certain level of heirarchical structure could be specified and developed as needed within that level of heirarchical structure prior to the need for developing other lower level components and structures. The fact that package and subprogram specifications and bodies can be separately compiled allows a tremendous amount of design freedom as well as a true simultaneous top-down-bottom-up design that enforces a modular and component discipline on the system designer and system implementer. It also allows a system designer to remain within the confines of the heirarchical level in which he was tasked to design without being overly concerned with the implementation details of a lower level upon which his heirarchical level will ultimately depend. An example of where separate compilability of program specifications and bodies can be

used is in the process of prototyping; where a simple and very high level system structure could be designed using program specifications as 'stubs' in place of the called program units. In this way the correctness and completeness of the initial design structure could be verified through compilation at a very early state in design and with a minimal investment of effort or time. Whether used in prototyping or other design strategies, the separate compilability feature of Ada is in direct support of the software engineering principles of abstraction and information hiding, and further directly supports heirarchical design methodologies.

As revealed in the General Electric/University of Maryland study, however, the Ada language in and of itself will not specifically prohibit non-conformance with a heirarchical design structure; it only encourages its use. The enforcement of heirarchical design must in the end be considered an essentially human endeavor, and perhaps the most valuable tool to use in this endeavor is that of structured walkthroughs during each phase of program or system development. Thus at any level of heirarchical design, the specifications of packages and subprograms within that level could be developed horizontally until the entire level was complete, after which that level could undergo the process of structured walkthroughs and specification compilation within that level. Only after it was

demonstrated that the level was complete and operable as to the various operations occurring within that level would the element of vertical development take place. That is to say, after it has been determined that the specifications within the present level of development are complete and operable, the program bodies belonging to each compiled program specification could then be developed. Of course once the vertical boundary between specification and body was crossed (a new level of heirarchical structure entered), the horizontal development requirement would have to be re-imposed within that new level, and the process of development, walkthroughs and compilation of program bodies (and specifications of even lower level program units called as a result of those bodies), would begin anew. This iterative process would continue in a horizontal-vertical-horizontal fashion until the entire system was complete. Regardless of whether Ada is utilized as a PDL or as an SDL, the need to employ this iterative, vertical-horizontal-vertical technique remains if the true value of Ada's features are to be realized in system design.

4. The Need for Support of Software Principles

In the design of any software system or of a PDL or SDL for utilization in the design of software systems, the fact remains that the software engineering principles of abstraction, information hiding, modularity, localization,

uniformity, completeness and confirmability must be maintained. Since it is people and not machines or languages that ultimately perform the process of software design, regardless of the programming or design language used, it is incumbent upon the people involved in system design, to ensure that these principles are continually enforced. As such, the enforcement of these principles is a management issue and not a language design issue.

The use of Ada as a PDL or SDL is not, after all, a design methodology in and of itself, but rather simply a method by which design can be represented. Put another way, Ada as a PDL is a concise and meaningful way to put down what is in the mind of the designer,but it will not by itself perform the design process. Ada will, however, greatly enhance the design process by virtue of the fact that it, more than any other language available, directly supports the software principles mentioned above.

D. LANGUAGE DESIGN ISSUES.

In addition to the management issues mentioned here, which are in effect applicable to the design of any software system, PDL or SDL, regardless of language implementation, there are specific language design issues which must be addressed prior to incorporating the Ada language into a specific SDL or PDL. At present there exists no single Ada based SDL or PDL as the accepted design within the DOD.

However, as mentioned in Chapter III, a number of ongoing development projects are underway under the auspices of DOD contract. While the projects differ in varying degrees as to language specifics, the approaches taken by the different vendors involved are quite similar. All, for instance, make use of the major Ada language features, though each makes use of those features at varying levels of implementation and effect.

In attempting to examine which Ada based PDL the DOD should ultimately adopt for use, it is appropriate to consider the necessary and desired features to include in that PDL, as well as the support environment within which the PDL will exist. The remainder of this chapter will be devoted to an examination of these features.

1. An Ada PDL Should be Applications Flexible

An Ada PDL should be versatile enough so as to allow its application at all levels of program design and development, regardless of program type or coding language. For example, if a design team is utilizing an Ada PDL in the design of a complex missile launch and guidance system, the use of the Ada PDL tool should not be constrained to any particular level or group of levels n the design heirarchy, but should instead be design-level independent. As such it should allow short-iteration prototyping at the highest levels of design while simultaneously supporting design of

the system's lowest level modules. Only with such design level independence will the Ada PDL allow true top-down-bottom-up system and module design.

2. An Ada PDL Should Not Subset the Ada Language

By this it is meant that an Ada based PDL should recognize the entire Ada syntax, rather than a subset of that syntax. The reason for this is that an Ada PDL which subsets the Ada language serves to restrict the available use of that language to the extent that the PDL is subsetted. In the same manner in which a person who relies solely on a pocket dictionary of the English language denies himself of many of the rich English words and constructs available in a more comprehensive dictionary, an Ada PDL which subsets the Ada language denies its users of some of the richness available in the Ada language.

3. An Ada PDL Should Include English Narratives

One major difference between most PDL's and high-level languages is that the PDL's use narrative text embedded directly within their statements. The purpose of this narrative text is to enhance understandability of both the individual statements and the design language as a whole. Another benefit is that since each PDL statement can be explicitly described to the user via a standardized narrative text format, there is far less confusion as to the purpose and intent of each PDL statement. The enforced

standardization serves to enhance the maintainability and transportability of both the PDL language itself, and the software systems and modules whose designs are a product of the PDL.

The use of English narratives serves also to encourage abstraction, the omission of implementation details where those details are unimportant to the present heirarchical level being designed. For example, if a prototype system were being designed using an Ada PDL, only the specifications to those program units being called by the prototype need be identified, with the associated program unit bodies being described only generally within the narrative text attached to the specifications. When the prototype was complete, it could then be compiled and checked for completeness and correctness as to the prototype alone and without regard for lower level implementation details. Once verified as complete and correct, those lower level details could be addressed, using the narrative text as a starting point. In this way, narrative text provides an effective vehicle with which to iteratively progress from a high level design to succeedingly more detailed program descriptions existing at lower levels in the system's heirarchy.

Unlike most other PDL's, an Ada based PDL allows the simple and safe inclusion of English narrative text without

effecting compilability of the PDL or its statements. As such, it is most important that an Ada PDL include extensive use of narrative text, and that that narrative text encourage abstract design level description rather than detailed coding.

4. An Ada PDL Should Allow Annotations

An Ada PDL should include a mechanism for expressing annotations which extend the Ada language in its application to a PDL. A primary purpose of annotations is to allow the expansion of Ada's KEYWORD dictionary so as to better match that dictionary to the particular needs of the PDL application. Annotations would best be used to provide additional design information or to impose requirements on the designer, such as specialized forms of Ada comments. It is not suggested that a specific set of annotations be adopted, but rather that a standard mechanism be identified to indicate annotations and that the use of annotations be encouraged in program design. The placement of specific annotations could be required to occur as a preamble to a design module or to precede particular PDL statements where appropriate. In addition to a standard means with which to express annotations within the PDL, a common Ada PDL processor which recognizes the annotation format and calls appropriate subroutines could be designed so as to ease the expandability of the annotation inventory.

There is of course a danger in the unchecked expansion of an Ada PDL through the use of annotations in that without some control, the PDL to which those annotations are added can become overly complex and ultimately unmanageable. Thus whereas annotations would provide a valuable tool in the flexibility of an Ada based PDL, their use should be judiciously controlled.

5. An Ada PDL Should be Supported by Automated Tools

One of the most valuable elements to be included in the Ada PDL environment would be that of an extensive automated tool inventory to assist in error-checking, design formatting and design editing. The Minimal Ada Programming Support Environment (MAPSE), as defined by the STONEMAN document, suggested that certain tools be included in the Ada language support inventory text editor, compiler and linker. As a minimum requirement to the Ada PDL support environment there should be an Ada compiler so as to ensure proper usage of the Ada language and minimize CPU time investment during program design. A PDL processor would be preferred over a compiler since it would allow the processing of annotations as well as the checking of errors in such a way as to report those errors in a manner compatible with design rather than implementation. Examples of errors that would be reported with a PDL processor are undefined subprograms or variables appearing in the program design. A third valuable tool would

be that of an Ada PDL text editor, where the purpose of such an editor would specifically be to encourage design rather than code in developing Ada PDL descriptions.

Another tool that could be of considerable value in the management of program design would be a graphics generator capable of generating data flow diagrams (DFD's), input-processing-output (IPO) charts, and flowcharts directly from the program being designed. While such a tool would perhaps need to be quite complex in order to perform the function of creating graphics directly from program code, the benefits possible from having automatically generated pictorial representations of programs under design are substantial. There is, for instance, perhaps no better way to convey the structure and purpose of a complex software program from one human being to another than through the vehicle of DFD's and IPO charts representing that program. An additional benefit of having an automatic graphics generator would be the ability to 'stand back' and view a graphical representation of the system under design at any desired point in time. Such a feature would further contribute to the proper placement of emphasis on the management of program design over the concern for implementation details, thereby helping to maintain control over the design process.

Regardless of which of the automated tools mentioned are developed for use with an Ada PDL, the Ada language is unique in its ability to support such tools once they are developed. The reasons for this lie primarily in Ada's strong typing and ability to support separate compilability of program unit specifications and bodies. Since all objects in Ada are explicitly defined and all interfaces specified, the task of creating a program structure and then checking that structure for completeness and correctness is significantly simplified, whether that task is performed manually or by machine.

6. An Ada PDL Should be Well Documented

Adequate documentation is a necessary element to any PDL, and an Ada based PDL is no exception. Documentation in the case of an Ada PDL should include a requirements document, an Ada PDL reference manual, an Ada PDL users guide and an Ada PDL processor users manual. These documents will be needed to establish a philosophy of Ada PDL usage and will be necessary for proper use of the Ada PDL. In addition they will serve to promote the wide-spread use of the Ada PDL for program design.

An additional form of documentation exists in the narrative text section of appropriate Ada statements, in that this narrative text serves in part to explain to the user the function and purpose of the associated Ada statement. In

this sense Ada is a self-documenting language, and the advantages possible through this self-documenting feature cannot be overstated. If, for example, a program designer using an Ada PDL was unsure as to the function of any particular statement within the PDL, he should be able to determine that function almost immediately simply by reading the narrative text attached to the statement. If he were still unsure, he could then consult the appropriate external publication, though such external consultation should be unnecessary if the tool of narrative text is exploited fully by the designers of the Ada PDL.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

Perhaps the single most important contributing factor to the present software crisis has been the failure on the part of software programmers and program designers to maintain a proper perspective on the need for management of the software projects they are tasked to accomplish. This failure is in part due to the fact that inadequate management style has been the rule rather than the exception, regardless of the software project involved or design language used; and in part due to the fact that, until Ada, there has been no language that tended to encourage a proper management style simply by virtue of language design.

The intent of this thesis has been first to explore the genesis of the present software crisis, to explore the various tools available in the Ada language which could be used to help avert future crises in software development, and lastly to address the applicability of the Ada language to its specific role as a program design language. It can be concluded from the points raised in this thesis that Ada is a far better candidate for implementation as a PDL than any other language presently in existence, and that the DOD

should place a very high priority on the design and implementation of the Ada language as a PDL.

Another important conclusion that can be drawn from the points raised in this thesis is that a definite shift in management policy and procedures must take place before the software crisis can be totally overcome. That is, management must shift its philosophy away from the perspective where piecemeal vertical software design and maintenance is tolerated; toward the perspective where comprehensive and horizontal design management is enforced.

B. RECOMMENDATIONS FOR FOLLOW-ON WORK

The authors recommend research be conducted at the Naval Postgraduate School in the following areas:

- Research the utility of implementing Ada as a language for use in software design at the Naval Postgraduate School.
- Utilization of Ada as a design language for use in embedded weapons systems such as Harpoon and Tomahawk.
- Continue research into use of Ada PDL as a compilable or non-compilable design language.

APPENDIX A

GLOSSARY

ABSTRACTION: The process of viewing a problem at a level of generalization where that level of generalization does not consider irrelevant lower level details. Abstraction can be likened to a "black box", where the person using or viewing the black box is concerned only with the functions of the black box as a whole and is not concerned with the elements making up the box. The use of abstraction allows one to view concepts and terms in the problem environment without having to transform them to the more detailed and less familiar solution environment.

ABSTRACT INTERFACE: Allows inputs into or outputs from a module to match changes in inputs or outputs so as only to effect the abstract interface code and not lower level code within the module.

ACCESS TYPE: A type whose objects are created by execution of an allocator. An access value designates such an object.

BODY: A program unit defining the execution of a subprogram, package or task. A body stub is a replacement for a body that is compiled separately.

BUBBLE DIAGRAM: See Data Flow Diagram (DFD).

CHARACTER: Any of the ASCII symbols that are used to form source Ada programs or are used as data. Graphic characters have a visible representation, while control characters have visible attributes that are implementation defined. Source programs are built from the graphic characters plus control characters which designate passage to a new line.

COLLECTION: The entire set of allocated objects of an access type.

CORRECTNESS: A program is correct if it performs properly the functions it was intended (specified) to do and has no unwanted side effects.

COMPILATION UNIT: A program unit presented for compilation as an independent text. It is preceded by a context specification which names the other compilation units on which it depends. A compilation unit may be the specification or body of a subprogram or package, including generic units or subunits.

CONTEXT SPECIFICATION: Prefixed to a compilation unit, defines the other compilation units upon which it depends.

CONVERSION: The process of translating from one type to another.

COUPLING: A measure of the relative independence among modules.

DATA FLOW DIAGRAM (DFD): A graphical tool used to depict data (information) flow within a system and between modules.

DECLARATION: Associates an identifier with a declared entity, including objects, types, subprograms, tasks, renamed entities, numbers, subtypes, packages, exceptions, and generic units.

DEBUG: The process of detecting and correcting errors in a procedure, system, process or module.

DECLARATIVE PART: A sequence of declarations and related information such as subprogram bodies and representation specifications that apply over a region of program text.

DERIVED TYPE: A type whose operations and values are taken from those of an existing type. The existing type is called the parent type.

DISCRETE TYPE: A type with an ordered set of distinct values. The discrete types are the enumeration and integer types. Discrete types may be used for indexing and iteration and for choices in case statements and record variants.

EMBEDDED COMPUTER SYSTEM: A computer system that forms part of a larger system whose purpose is not primarily computational, such as a weapons system or process controller.

EMBEDDED PROGRAM: A computer program that is part of some larger entity and essential to the proper operation of that entity. For example, the program which serves to

identify different aircraft in flight is embedded within the air traffic control system.

ENTITY: Anything that can be named or denoted in a program. Objects, types, values and program units are all entities.

ENTRY: Used for communication between tasks. Externally, an entry is called just as a subprogram is called.

ENUMERATION TYPE: A discrete type whose values are given explicitly in the type declaration. These values may be either identifiers or character literals which are considered enumeration literals.

EXCEPTION: An event that causes suspension of normal program execution. An exception handler is a piece of program text which specifies a response to the exception and the execution of such a program text is called handling the exception.

EXECUTE: To carry out an instruction or to perform a routine or set of routines.

EXPRESSION: Part of a program that computes a value.

FLOWCHART: A graphical tool used to show sequence and control of program or module logic.

FUNCTION: The name given to one or more statements that perform a specific task.

GENERIC PROGRAM UNIT: A subprogram or package specified with a generic part. A generic clause contains the declaration of generic parameters, which may be types, subprograms or objects. In the generic specification these are called generic formal parameters. When the unit is instantiated the formal parameters are matched with the actual parameters. A generic program unit may be thought of as a possibly parameterized model of program units. Instantiated program units define subprograms and packages that can be used directly in a program.

IDENTIFIER: One of the basic lexical elements of the language. An identifier is used as the name of an entity or a reserved word.

INFORMATION HIDING: Specification and design of modules such that information (procedures and data) contained within a module are inaccessible to other modules which have no need to know the information.

INTERFACE: Communication between modules governed by a set of assumptions one module makes about another.

LEXICAL UNIT: One of the basic syntactical elements making up a program. A lexical unit is an identifier, a number, a character literal, a string, a delimiter or a comment.

LIBRARY UNIT: A compilation unit that is not a subunit of another unit and which belong to a program library.

MAINTENANCE: The phase in a system's life cycle following development, acceptance and installation.

MODULE: A separately addressable element within a program.

MODULAR DESIGN: A logical partitioning of software into elements that perform specific functions or subfunctions.

OBJECT: A variable or constant. An object can denote any kind of data element, whether a scalar value, a composite value, or a value in access type.

PACKAGE: A program unit specifying a collection of related entities such as constants, variables, types, and subprograms. The visible part of a package contains the entities which may be used from outside the package; while the private part contains structural details that are irrelevant to the user of the package but complete the specification of the visible entities. The package body contains the implementation of subprograms or tasks (possibly other packages) specified in the visible part,

PARAMETER: One of the named entities associated with a subprogram, entry, or generic program unit. A formal parameter is an identifier used to denote the named entity in the unit body. An actual parameter is the particular entity associated with the corresponding formal parameter in a subprogram call, entry call, or generic instantiation.

PRIVATE TYPE: A type whose structure and set of values are clearly defined but not known to the user of the type. A private type and its applicable operations are defined in the visible part of the package.

PROGRAM LIBRARY: Part of the Ada program support environment data base recognized by the Ada compiler, consisting of a collection of compilation units.

PROGRAM UNIT: Any of the three primary structures making up an Ada system; namely, subprograms, packages and tasks.

RANGE: A contiguous set of values of scalar type. A range is specified by giving the lower and upper bounds for the values.

RENDEZVOUS: The interaction that occurs between two parallel tasks when one task has called the entry of the other task and a corresponding accept statement is being executed by the other task on behalf of the calling task.

REQUIREMENTS ANALYSIS: The third step in the software engineering procedure and the last step of the planning phase. Describes the software by identifying the interface details and an in-depth description of functions; determining design constraints and defining software validation requirements.

ROBUSTNESS: The ability of a program or software system to handle unforeseen environmental changes (such as hardware

failure) and demands (such as data) in a "graceful" or reasonable fashion.

SCALAR TYPES: A type whose values have no components. Scalar types comprise discrete types (enumeration and integer types) and real types.

SOFTWARE ENGINEERING: Software implementation of a problem solution approached by using a set of techniques that are application independent. These techniques are: (1) a well-defined methodology that addresses a software life cycle of planning, development and maintenance; (2) an established set of software components that documents each step in the life cycle and shows traceability from step to step; and (3) a set of predictable milestones that can be reviewed at regular intervals throughout the software life cycle.

SOFTWARE PLAN: The second step in the software engineering process. Provides a framework enabling the manager to make reasonable estimates of resources, cost and schedule.

STATEMENT: As opposed to a declaration which defines an entity, the execution of a statement causes some action to be performed.

SUBPROGRAM: An executable program unit, possibly with parameters for communication with its point of call. A subprogram declaration specifies the name of the subprogram and its parameters; a subprogram body specifies its

execution. A subprogram may be a procedure which performs an action or a function which returns a result.

SYSTEM: A collection of elements related in a way that allows accomplishment of some tangible objective.

SYSTEM DEFINITION: First step in the software planning phase where attention is focused on the system as a whole. Functions are allocated as to hardware, software, and other system elements based on a preliminary understanding of system requirements.

TASK: A program unit that may operate in parallel with other program units. A task specification establishes the name of the task and the names and parameters of its entities, while a task body defines its execution. A task type is a specification that permits the subsequent declaration of any number of similar tasks. A task is said to depend upon the unit in which it is declared (subprogram body, task body or library package body). A unit is not left until dependent tasks are terminated. A task is completed if it is waiting at the end of its body for any dependent tasks or is aborted but not yet terminated. A completed task cannot be called. A terminated task is, in a sense, the same as a dead task (it is no longer active).

TYPE: Characterizes a set of values and a set of operations applicable to those values. A type definition is a language construct introducing a new, unique type, whereas a

subtype creates a compatible (possibly) constrained definition of the base type. A type declaration associates a name with a type introduced by a type definition.

VISIBILITY: At a given point in the program text, he declaration of an entity with a certain identifier is said to be visible if the entity has an acceptable meaning for an occurrance at the point of the identifier.

APPENDIX B

ACRONYMS AND ABBREVIATIONS

APSE	Ada Program Support Environment
DARPA	Defense Advanced Research Projects Agency
DFD	Data Flow Diagram
HARRIS/PDL	Process Description Language
HOLWG	High Order Language Working Group
KAPSE	Kernel Ada Program Support Environment
MAPSE	Minimal Ada Program Support Environment
ONR	Office of Naval Research
PDL	Program Design Language
RFP	Request for Proposal
SDL	System Design Language

APPENDIX C

PDL VS. ADA COMPARISON

Language Features	HARRIS	TRW	IBM	NORDEN
Commentary:				
Comments:	X	-	X	X
Pragma:	-	X	-	-
Declarations:				
Types				
Scalar Types:	X	X	X	X
Array Types:	X	X	X	X
Record Types	X	X	X	X
Access Types:	X	X	X	X
Private Types:	X	X	X	X
Derived Types:	X	X	X	-
Sub Types:	X	X	X	X
Limited Private:	X	X	X	X
Object Declarations:				
Simple Declarations:	X	X	X	X
Array Declarations:	X	X	X	X
Discriminants and Variants:				
Names & Expressions				
Attributes:	X	X	X	X
Slices:	X	X	X	X
Aggregates:	X	X	X	X
Expressions:	X	X	X	X
Type Conversions:	X	-	X	-
Qualified Expressions:	X	X	X	X
Overloading Opertors:	X	X	X	X
Allocators:	X	X	-	X
Statements:				
Label:	X	X	-	X
Assignment:	X	-	X	X
If (ELSE-ELSIF):	X	X	P	X
Case:	X	X	X	X
Loop:	X	X	P	X
While:	X	X	X	X
For:	X	X	X	X
Blocks:	P	X	X	X

	HARRIS	TRW	IBM	NORDEN
Exit:	X	X	P	X
Return:	X	X	X	X
Goto:	-	X	-	X
Null:	X	-	X	X
Subprograms:				
Procedures:	X	X	X	X
Procedure Call:	X	-	X	X
Functions:	X	X	X	X
Function Call:	X	-	X	X
Positional Parameters:	X	-	X	X
Named Parameters:	X	-	X	X
Packages:				
Specification:	X	X	X	X
Body:	X	X	X	X
Visibility:				
Use:	X	X	X	X
Rename:	-	X	-	-
Tasking:				
Task Types:	X	X	-	X
Entry:	X	X	-	X
Accept:	X	X	-	X
Delay:	X	X	-	X
Selective Wait:	X	X	-	X
Conditional Entry Call:	X	X	-	X
Timed Entry Call:	X	X	-	X
Abort:	X	-	-	X
Program Structure:				
Compilation Units:	X	X	X	X
Subunits:	X	X	X	X
With:	X	X	X	X
Exceptions:				
Handlers:	X	X	-	X
Raise:	X	-	-	X
Suppress Pragma:	-	-	-	-
Generics:				
Subprogram:	X	X	-	X
Packages:	X	X	X	X
Instantiation:	X	X	X	X

	HARRIS	TRW	IBM	NORDEN
Representation:				
Length:	X	-	-	-
Enumeration:	X	-	-	-
Record:	X	-	-	-
Address:	X	-	-	-
Machine Code Insertion:	X	-	-	-
I/O:				
Package Input-Output:	X	-	-	-
Package Text-IO:	X	-	-	-
Get:	X	-	-	-
Put:	X	-	-	-
Read:	X	-	-	-
Write:	X	-	-	-
Package Low-level-ID:	X	-	-	-
Send-Control:	X	-	-	-
Receive-Control:	X	-	-	-

X : Supported
- : Not Supported
P : Partially Supported

LIST OF REFERENCES

1. Boehm, Barry W., "Software and Its Impact: A Quantitative Assessment", Datamation, p. 5-16, May 1973.

2. Carlson, W. E., Druffel, L. F., Fisher, D. A., and Whitakker, W. A., Introducing Ada, Paper presented at the Annual Conference of the Association for Computere Machinery, 27-29 October 1980.

3. Pressman, Roger S., Software Engineering: A Practitioner's Approach, McGraw-Hill Book Company, 1982.

4. Ross, D. T., Goodewough, J. B., and Irvine, C. A., "Software Engineering: Process, Principles and Goals", Computer, p. 54-64, May 1975.

5. Stevens, W. P., Myers, G. J., and Constantine, L. L., "Structured Design", IBM Systems Journal, v. 13, No. 2, p. 115-138, 1974.

6. Yourdon, E., and Constantine, L., Structured Design: Fundamentals of a Discipline of Computer Program and System Design, Prentice-Hall, 1979.

7. Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, v. 15, No. 12, p. 220-225, December, 1972.

8. Booch, G., Software Engineering with Ada, Benjamin/Cummings Publishing Company, 1983.

9. Wegner, P. W., The Ada Programming Language and Environment, Unpublished Report, 6 June 1981.

10. Softech, Inc., Ada Programming Design Language Survey, by L. Lindley and R. Sheffield, October 1982.

11. Caine, S. and Gordon, E., PDL--A Tool for Software Design, Proceedings, National Computer Conference, 1975.

12. Basili, V., and Others, "Monitoring an Ada Software Development", Office of Naval Research Newsletter, v. 1, No. 2, December 1982.

13. Wegner, P. W., "Ada Education and Technology Transfer Activities", Ada Letters, v. 2, No. 2, 1982.

14. Freedman, R. S., _Programming Concepts with the Ada Language_, Petrocalli Books, Inc., 1982.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
Cameron Station
Alexandria, Virginia 22314

2. Library, Code 0142 2
Naval Postgraduate School
Monterey, California 93940

3. Department Chairman, Code 54 1
Department of Administrative Science
Naval Postgraduate School
Monterey, California 93940

4. LCDR George J. Wylie, USN 1
Patrol Squadron Nineteen
FPO San Francisco, CA 96601

5. LT Thomas R. Watt, USN 1
Commander Amphibious Squadron Eight
FPO New York, NY 09501

6. LCDR Ronald Modes, USN, Code 52MF 2
Department of Computer Science
Naval Postgraduate School
Monterey, California 93940

7. LCDR John R. Hayes, SC, USN, Code 54HT 1
Department of Administrative Science
Naval Postgraduate School
Monterey, California 93940

8. Naval Postgraduate School 1
Computer Technologies Curricular Office
Code 37
Monterey, California 93940

201696

Thesis
W955 Wylie
c.1 Utilization of Ada
as a program design
language.